VŠB – Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky

BAKALÁŘSKÁ PRÁCE

2010 Jaroslav Dyba

VŠB – Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Implementace algoritmu Paige-Tarjan

Implementation of the algorithm Paige-Tarjan

2010 Jaroslav Dyba

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

V Ostravě 24.4.2010 ……..

Rád bych poděkoval vedoucímu mé bakalářské práce Ing. Zdeňku Sawovi, Ph.D. za jeho podnětné návrhy a čas, který mi věnoval.

Abstrakt

Práce se zaobírá vysvětlením algoritmu Paige-Tarjan, jeho implementací a otestováním na konečně stavových systémech. Algoritmus bude implementován v programu Paige–Tarjan napsaném v programovacím jazyce Java. Program zanalyzuje zadaný vstupní systém, přetransformuje jej a změří efektivnost celého procesu. Vstupní systém může být definován přímo v aplikaci nebo nahrán z externího souboru. Grafické uživatelské rozhraní bude implementováno pomocí knihovny Swing. Pro ověření efektivnosti algoritmu budou použity systémy s různou vnitřní strukturou.

Klíčová slova: Algoritmus, Paige-Tarjan, verifikace, implementace, Java

Abstract

The main objectives of this thesis is to implement Paige – Tarjan algorithm in programming language Java, verify the efficiency of this algorithm and test it on finite state systems. The Paige – Tarjan application is created to verify the efficiency of this algorithm. The program analyse given system, transform it and measure the efficiency of the whole process. Input systém could be loaded from external file or directly defined in the application. Graphical user interface is constructed with help of Swing GUI Components. The solution will be tested by using input systems with different structure.

Keywords: Algorithm, Paige-Tarjan, verification, implementation, Java

Obsah

1 Úvod

V životě se můžeme setkat se situací, kdy potřebujeme srovnat chování dvou různých systémů. Vždy záleží, z jakého pohledu systémy porovnáváme. Srovnání chování dvou jednoduchých systémů jako např. dvou různých automatů může být triviální, ale pro srovnání mnohem složitějších a komplexnějších systémů, jako např. elektrické rozvodné sítě a vodovodní sítě již potřebujeme mnohem sofistikovanější postupy.

Jednou z možností verifikace systému je testování ekvivalence chování dvou systémů. Formální vyjádření stejného chování dvou systémů je bisimulační ekvivalence. V teoretické informatice je bisimulace relací ekvivalence mezi stavy přechodových systémů. Bisimulační ekvivalenci si můžeme představit jako schopnost dvou systémů navzájem si dorovnat své tahy. Z vnějšího pohledu nám chování dvou bisimilárních systémů připadá stejné.

Cílem mé práce je naimplementovat algoritmus Paige-Tarjan, který je schopen rozhodnout, jestli chování dvou systému je v nějakém smyslu stejné a tento algoritmus otestovat. Testování algoritmu provedu na konečně stavových systémech. Abych mohl testovat algoritmus na velkých konečně stavových systémech, tak budu muset vytvořit náhodný generátor vstupu. Program bude implementován v programovacím jazyce Java.

2 Popis algoritmu Paige - Tarjan

V této kapitole nastíníme problém a vysvětlíme si tzv. bisimulační ekvivalenci, která je s řešením tohoto problému úzce spjata. Popíšeme si kroky vedoucí ke zjednodušení vstupního systému a ukážeme si jádro algoritmu, které se skládá ze sedmi kroků.

2.1 Problém hledání nejhrubšího relačního rozkladu

Mějme systém tvořený ze stavů a přechodů mezi nimi. Stavy budou realizovány *uzly* a podmínky přechodu mezi jednotlivými stavy budou reprezentovány *hranami*. Na začátku má systém prvotní rozklad, čili jeho stavy jsou rozděleny do množin. Těmto množinám stavů budeme říkat *bloky.*

Problémem je, jak vstupní systém rozložit na konečný rozklad. V konečném rozkladu jsou všechny bloky mezi sebou navzájem *stabilní*. Blok A je stabilní vůči bloku B, pokud všem uzlům z bloku A vede hrana do bloku B, popřípadě z bloku A nevede žádná hrana do bloku B.

Konečného rozkladu systému můžeme docílit tak, že použijeme „přímý“ algoritmus. Přímý algoritmus je velice naivní. Pro zjemnění systému vybere postupně všechny bloky. Pokud se mu již vybraný blok rozpadl, musí provést ještě rozklad podle obou nově vzniklých bloků. Aktuálně vybraný blok, podle kterého se budou rozpadat ostatní bloky, se nazývá *splitr*.

Robert Paige a Robert E. Tarjan řešili problém konečného rozkladu systému. Jejich algoritmus je založený na kombinaci Hopcroftové myšlenky „výběru menší poloviny“ a rozpadu bloku na tři části. Výběr splitru podle Hopcroftovy myšlenky spočívá v tom, že ze *skupiny* vybereme blok, který neobsahuje více než polovinu uzlů celé skupiny. *Skupina* je množina bloků. U přímého algoritmu se bloky rozpadaly tak, že z původního bloku se vybraly uzly, které měli hrany směřující do splitru a vytvořil se pro ně nový blok. Rozpad bloku na tři části spočívá v tom, že z nového bloku, ve kterém jsou uzly s hranami směřující do splitru, vybereme uzly, které mají pouze hrany do splitru. Nyní se blok rozpadl na tři bloky. V původním bloku zůstaly uzly, které nemají hranu směřující do splitru. Ve druhém bloku jsou uzly, které mají hrany směřující jak do splitru, tak i do jiných bloků a ve třetím bloku jsou uzly s hranami směřující pouze do splitru. Díky těmto úpravám má algoritmus Paige-Tarjan časovou složitost O (m*log*n), kde m je množina přechodů a n je množina stavů a paměťová náročnost je O (m+n).

Pokud porovnáváme dva systémy, jestli mají podobné chování, tak nejdříve provedeme jejich disjunktní sjednocení. Vstupní stavy obou systémů budou obsaženy ve společném bloku. V konečném rozkladu jsou uzly obsažené ve stejném bloku ve vzájemné bisimulační ekvivalenci. Pokud původní vstupní stavy obou systémů jsou v konečném rozkladu ve stejném bloku, tak oba systémy mají podobné chování.

2.2 Bisimulační ekvivalence

Pojem bisimulační ekvivalence zavedl Park [3].

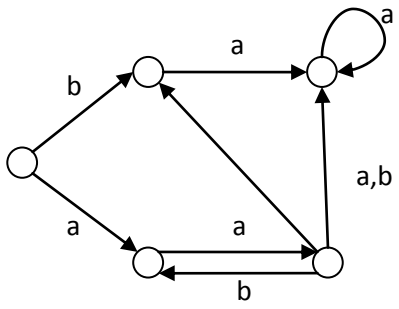
Přechodový systém je trojice T = (S,A, $\rightarrow$) kde:

S – množina stavů,

A – konečná množina akcí,

$\rightarrow \subseteq$ S × A × S - přechodová relace

Přechodovou relaci můžeme zapsat jako: $s \xrightarrow{a} s'$



Obrázek 2.1. Přechodový systém

Binární relace R na množině stavů přechodového systému je bisimulací, jestliže pro libovolnou dvojici stavů (s, t) ∈ R platí:

Pokud $s \xrightarrow{a} s'$ pro nějaké a a s′, pak $t \xrightarrow{a}$ t′ pro nějaké t′ takové, že (s′, t′) ∈ R.

Pokud $t \xrightarrow{a} t'$ pro nějaké a a t′, pak $s \xrightarrow{a} s'$ pro nějaké s′ takové, že (s′, t′) ∈ R.

Definice

Máme relaci, která je podmnožinou SxS

Relace ~ taková, že s ~ t, právě když existuje nějaká bisimulace R taková, že (s,t) ∈ R, se nazývá bisimulační ekvivalence neboli bisimilarita.

Bisimulační ekvivalence jako hra

Bisimulační ekvivalenci jako hru vysvětluje Colin Stirling[2], vykládá ji jako hru G (s_0,t_0), kterou hrají dva hráči, hráč 1 a hráč 2. Hra se odehrává na konečné nebo i nekonečné množině stavů (s_0,t_0)...(s_i,t_i). První hráč chce dokázat, že počáteční stavy jsou odlišné a druhý hráč chce dokázat opak, že jsou počáteční stavy rovnocenné. Předpokládejme, že počáteční stav hry je (s_0,t_0)...(s_j,t_j). Do následující dvojice stavů (s_{j+1},t_{j+1}) je možné se dostat dvěma způsoby.

-Hráč 1 si vybere přechod $s_j \xrightarrow{a} s_{j+1}$ a potom hráč 2 vybere přechod se stejným pojmenováním přechodu $t_j \xrightarrow{a} t_{j+1}$.

-Hráč 1 si vybere přechod $t_j \xrightarrow{a} t_{j+1}$ a potom hráč 2 vybere přechod se stejným pojmenováním přechodu $s_j \xrightarrow{a} s_{j+1}$.

Hra pokračuje dalšími tahy. Hráč 1 vždy vybírá tah jako první, a potom táhne hráč 2, který musí dodržet tah se stejným pojmenováním přechodu jako hráč 1.

Hra končí v případě, že v pozici (s,t) má jeden z hráčů přechod *a* a druhý nikoliv, pak jsou systémy *s* a *t* rozdílné. Proto v jakékoliv pozici (s_n,t_n), kde s_n a t_n jsou rozdílné, vyhrává hráč 1 a říkáme, že v této pozici má vítěznou strategii. Pokud se hráči 1 nepodaří nález pozici, ve které má vítěznou strategii, tak vyhrává hráč 2 a říkáme, že vítěznou strategii má hráč 2. Vítězná strategie hráče 2 nastane, když mohou táhnout oba hráči do nekonečna nebo se dostanou do stavu, odkud nevede žádný možný tah.

Různé způsoby hry mohou mít různé vítěze. Avšak, pro každou hru může vyhrát jen jeden z hráčů, bez ohledu na to, jak hraje jeho soupeř. Tahy tohoto hráče musí být přesné, proto je strategie velmi důležitá. Strategie je soubor pravidel, které říkají hráči, jak má táhnout. Následující pravidla nejsou závislá na předchozích tazích.

Pravidlo pro hráče 1 je: „ve stavu (s,t) vyber přechod *x*“, kde x je $s_j \xrightarrow{a} s_{j'}$ nebo $t_j \xrightarrow{a} t_{j'}$ pro nějaký přechod *a*.

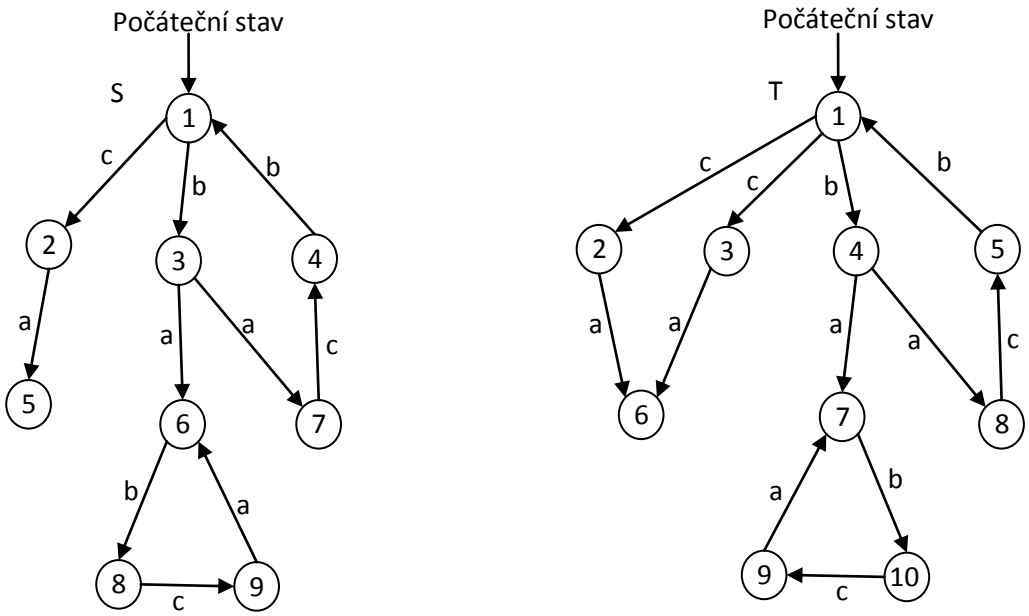
Pravidlo pro hráče 2 je: „ve stavu (s,t), když hráč 1 vybere *x*“, kde x je buď $s_j \xrightarrow{a} s_{j'}$ nebo $t_j \xrightarrow{a} t_{j'}$ a *y* je ekvivalentní přechod ve druhém systému.

Pokud hráč použije svou strategii a v ní dodrží výše zmíněné pravidla a vyhraje každou hru, potom říkáme, že má vítěznou *strategii*.

Pokud má vítěznou strategii hráč 1, potom se mu povedlo dokázat, že systémy nejsou ekvivalentní.

Pokud má vítěznou strategii hráč 2, potom se mu povedlo dokázat, že systémy jsou ekvivalentní.

Vysvětlení bisimulační ekvivalence jako hry na příkladu



Obrázek 2.2. Ukázka dvou systémů

Na Obrázek 2.2 máme dva systémy, u kterých chceme ověřit, jestli mají shodné chování. První systém nazveme S a druhý T. Oba mají jeden počáteční stav a to v uzlu číslo 1. První hráč vybírá systém, ve kterém provede tah, druhý hráč se snaží provést stejný tah, ale ve druhém systému.

Příklady her:

1. hra:

1. kolo: Hráč 1 vybere systém a provede tah: T - $1 \xrightarrow{c} 3$, hráč 2 provede stejný tah ve druhém systému: S - $1 \xrightarrow{c} 2$.

2. kolo: Na tahu je hráč 1: S - $2 \xrightarrow{a} 5$, hráč 2 táhne: T - $3 \xrightarrow{a} 6$, v tuto chvíli hra končí, protože hráč číslo 1 nemá žádný možný tah k provedení ani v jednom ze systémů.

Vyhrává hráč číslo 2. Všimněme si, že pokud by první hráč zvolil tah: T - $1 \xrightarrow{c} 2$ místo tahu: T - $1 \xrightarrow{c} 3$, hra by dopadla stejně. Stavy 2 a 3 v systému T jsou ekvivalentní. Když se na ně podíváme, tak mají stejné chování a mohly by se nahradit jediným stavem.

2. hra:

1. kolo: Hráč 1 táhne: S - $1 \xrightarrow{b} 3$, hráč 2 má jen jeden možný tah: T - $1 \xrightarrow{b} 4$.

2. kolo: Další tah hráče 1 je: S - $3 \xrightarrow{a} 6$, teď si hráč 2 může vybrat, jestli pojede do stavu 5, 7 nebo 8. Vybere třeba tento tah T - $4 \xrightarrow{a} 7$.

Hra se teď nachází v situaci, kdy každý tah hráče1, dokáže hráč 2 následovat ve druhém systému. Hra se zacyklila a mohla by se hrát do nekonečna, v tomto případě opět vyhrává hráč 2.

3. hra:

1. kolo: Hráč 1 táhne: T - $1 \xrightarrow{b} 4$, hráč 2 má jen jeden možný tah: S - $1 \xrightarrow{b} 3$.

2. kolo: Další tah hráče 1 je: T - $4 \xrightarrow{a} 5$, teď má hráč 2 dva tahy, ze kterých si může vybrat. Vybere tah: S - $3 \xrightarrow{a} 7$.

3. kolo: Hráč 1 táhne: S - $7 \xrightarrow{c} 4$, v tuto chvíli hráč 2 nedokáže provést stejný tah ve druhém systému a vyhrává hráč 1. Vraťme se do 2. kola, kde si hráč 2 vybíral ze dvou tahů.

2. kolo: Hráč jedna hrál tah: T - $4 \xrightarrow{a} 5$, hráč dvě teď zvolí druhý možný tah: S = $3 \xrightarrow{a} 6$. V následujících 3 kolech jsou tahy vynucené a není kde se rozhodovat. Zajímavější je ovšem 6. kolo, kde se nachází oba systémy ve stavech 6 a na tahu je hráč 1. V systému S není z tohoto stavu žádný možný tah, proto hráč 1 provede tah v systému T. V této situaci nedokáže hráč 2 napodobit jeho tah a vyhrává hráč 1. Hráč 1 má vítěznou strategii, protože když se dostane do stavu T - 4 a provede tah T - $4 \xrightarrow{a} 5$, poté ať udělá hráč 2 jakýkoliv tah, tak následně prohraje.

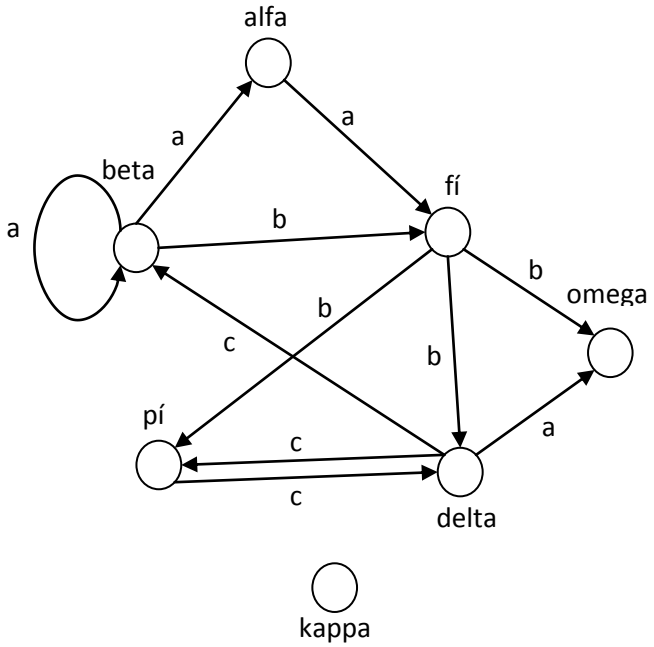
Závěr z her:

Systémy nejsou ekvivalentní, protože hráč 1 má vítěznou strategii a to i přes to, že v našem příkladě první dvě hry vyhrál hráč 2. Systémy by si byly ekvivalentní, pokud by systém T neměl přechod ze stavu 4 do stavu 5.

2.3 Kroky vedoucí ke zjednodušení systému

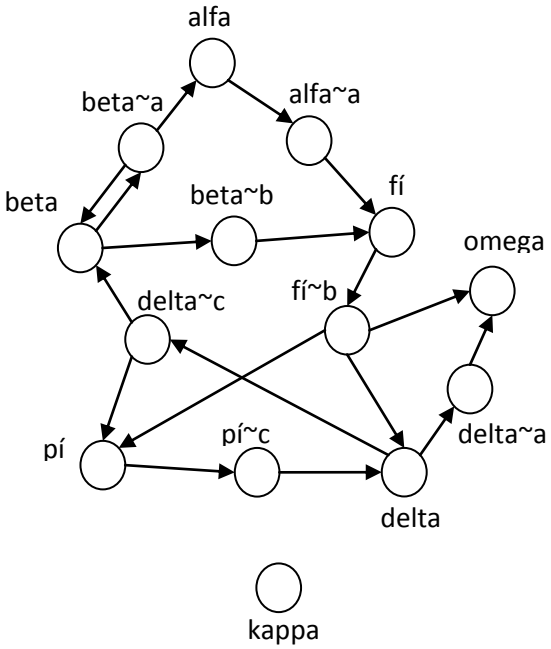
V následujících částí se už budeme zaobírat samotným algoritmem Paige-Tarjan. Protože vstupní systém je většinou příliš složitý, snažíme se ho zjednodušit a rozložit na menší části. Rovněž detailnější popis systému nám později pomůže při aplikaci samotného algoritmu.

Zjednodušení vstupu



Obrázek 2.3. Ohodnocený orientovaný systém

Vstupem může být ohodnocený přechodový systém. Tento systém lze reprezentovat pomocí orientovaného ohodnoceného grafu znázorněného viz Obrázek 2.3. Tento graf obsahuje pojmenované přechody, které můžeme odstranit a graf tím zjednodušit. Pojmenování přechodu odstraníme tím, že mezi dva uzly spojené hranou vložíme uzel a původní hrana se rozpadne na dvě hrany. Pokud uzel měl více přechodů se stejným pojmenováním, tak pro ně vznikne jen jeden společný uzel a z něj povedou hrany do ostatních uzlů. Na Obrázek 2.4 je znázorněn orientovaný graf, který jsme získali zjednodušením původního grafu.



Obrázek 2.4. Orientovaný systém

Vytvoření hran směřujících do předchůdců uzlu

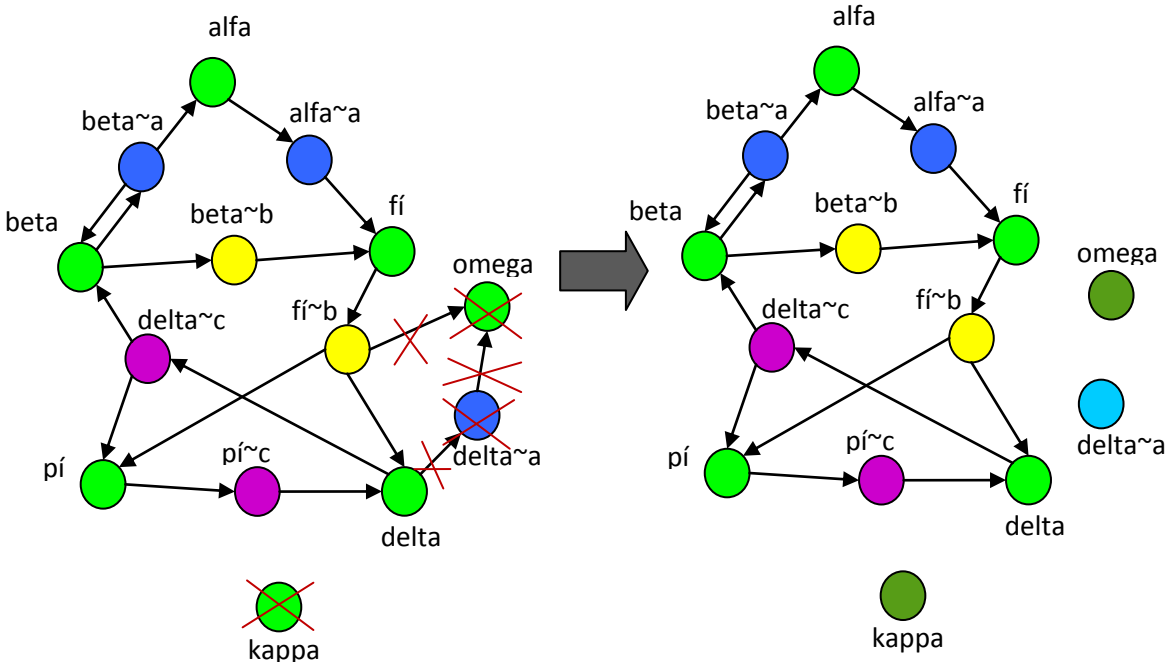
Pro operace, které bude algoritmus provádět na systému, potřebujeme znát pro každý uzel jeho *předchůdce*. Předchůdce je uzel, který má hranu vedoucí do aktuálního uzlu. Ty ale nemáme zadané, známe pouze hrany vedoucí do následníků uzlů. U každého uzlu můžeme vytvořit seznam hran, přes které budeme získávat předchůdce. Tento seznam vytvoříme tak, že vezmeme postupně všechny uzly a budeme procházet jejich hrany vedoucí do následníků. Následníkovi vložíme aktuální hranu do seznamu.

Předzpracování

Před samotným spuštěním algoritmu můžeme provést ještě rozpad vstupního systému. V blocích může být uzel, popřípadě uzly, které nemají žádnou hranu, která vede do jiného uzlu. Po skončení algoritmu by tyto uzly skončili v samotném bloku. My můžeme ještě před samotným spuštěním tyto uzly vyhledat a bloky ve kterých se nachází rozložit.

Pokud blok takový uzel obsahuje, tak pro něj vytvoříme nový blok a přesuneme ho do tohoto nového bloku. V původním bloku se může nacházet více takových to uzlů. Všechny tyto uzly vložíme do jednoho bloku. Po tomto předzpracování se nám počet bloků maximálně zdvojnásobí, a to v případě, pokud by se takový to uzel vyskytoval v každém bloku.

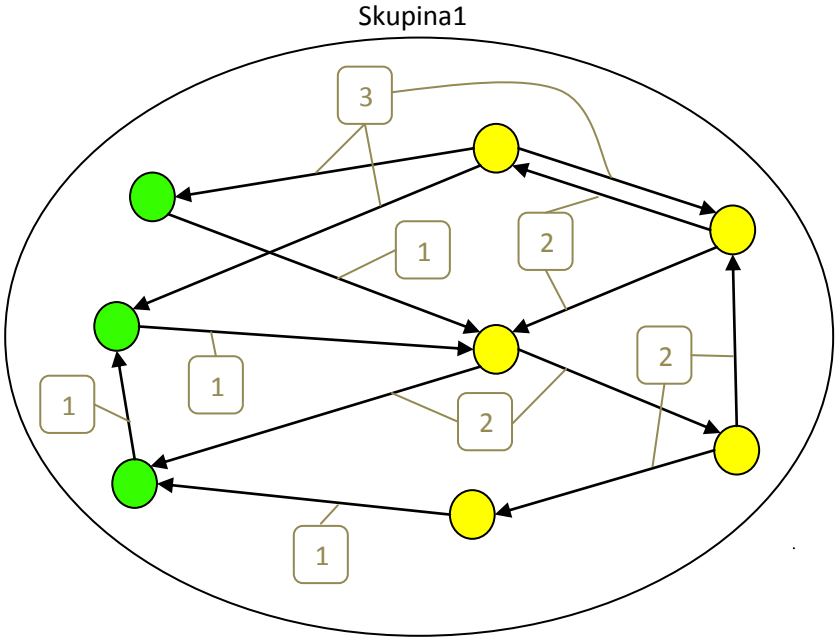
V našem systému vytvoříme nový blok pro uzel s názvem *kappa* a *omega*, protože nemají žádnou hranu vedoucí do jiného uzlu. Vytvoříme pro ně blok a vložíme je společně do něj, protože před rozpadem patřily do stejného bloku. Musíme odstranit všechny hrany vedoucí do těchto uzlů. Odstraníme hrany vedoucí do uzlu s názvem *omega*. Po odstranění hran máme uzel s názvem *delta~a*, ze kterého nevede žádná hrana. Tento uzel vyjmeme a vytvoříme pro něj nový blok. Odstraníme všechny hrany, které do něj vedly. Nyní předzpracování skončí, protože už všechny uzly mají hrany vedoucí do jiného bloku. Předzpracování je znázorněno viz Obrázek 2.5.



Obrázek 2.5. Znázornění předzpracování

2.4 Jádro algoritmu

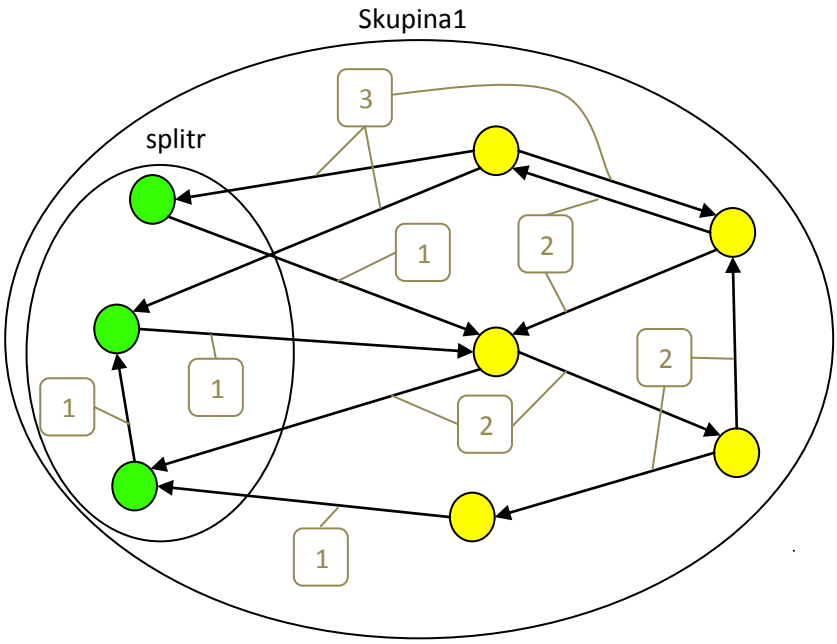
Samotné kroky algoritmu jsou popsány v článku od Roberta Paigeho a Roberta E. Tarjana [1]. Uvedeme si jednoduchý příklad systému, na kterém si popíšeme kroky algoritmu. Systém je rozdělený do dvou bloků a všechny bloky jsou v jedné skupině. Máme vytvořené čítače a každá hrana má v sobě referenci na nějaký čítač. U uzlu jsou vytvořeny všechny hrany a každý uzel má v sobě čítač.



Obrázek 2.6. Vstupní systém

Krok 1 Výběr splitru

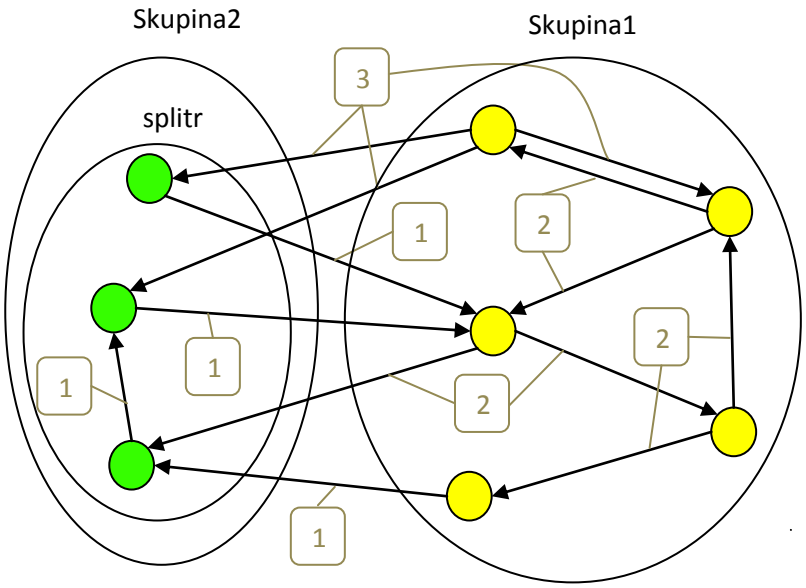
Vybereme skupinu ze spojitého seznamu. Z této skupiny vybereme blok, který nesmí mít více uzlů, než je polovina všech uzlů ve skupině. To provedeme tak, že získáme první dva bloky a porovnáme jejich počty uzlů mezi sebou, menší z těchto dvou bude spliter.



Obrázek 2.7. Výběr splitru

Krok 2 Rozpad skupiny

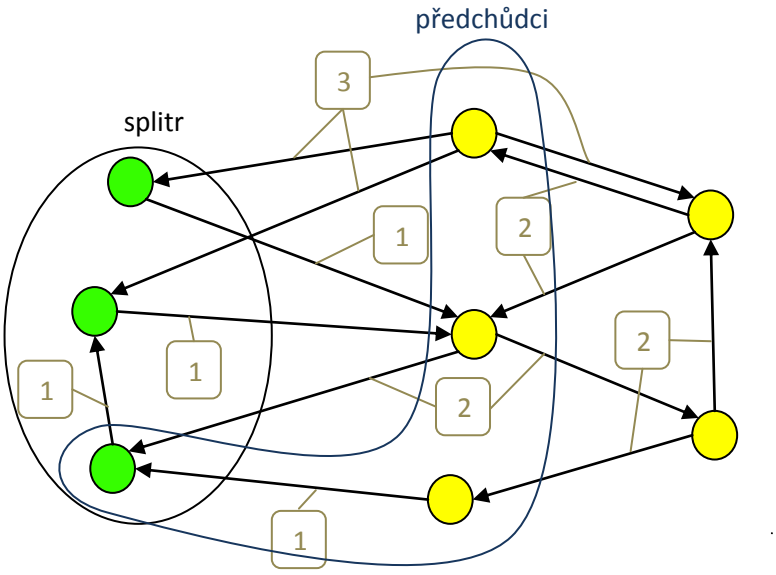
Vybraný spliter vyjmeme ze skupiny. Vytvoříme pro něj novou skupinu a vložíme ho do ní. Pokud původní skupina obsahuje minimálně dva bloky, tak tuto skupinu vložíme zpátky do spojitého seznamu.



Obrázek 2.8. Rozpad skupiny

Krok 3 Vytvoření předchůdců bloku

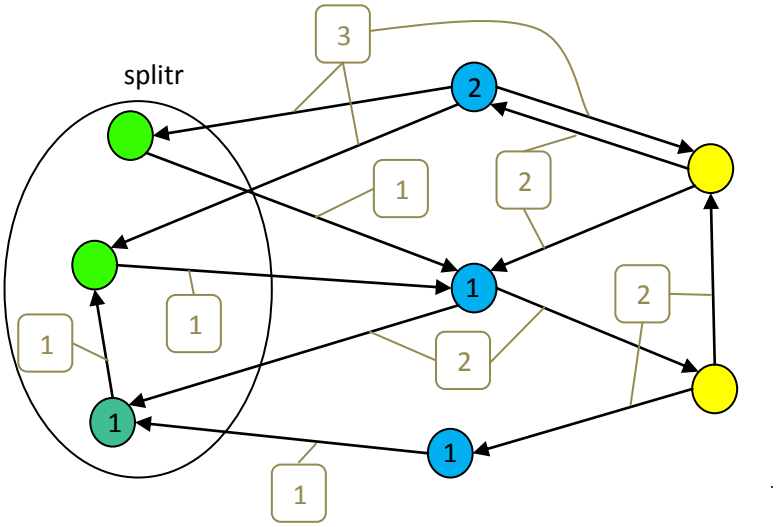
Vytvoříme předchůdce splitru. Od každého uzlu ve splitru získáme hrany, které vedou do aktuálního uzlu. Z těchto hran získáme přechůdce a uložíme si je do spojitého seznamu předchůdců bloku. Duplicitu budeme řešit tím, že při každém přidání uzlu do spojitého seznamu, nastavíme u uzlu příznak, že už byl uzel přidán do seznamu. Ještě před každým přidáním budeme tento příznak testovat, a pokud příznak signalizuje, že už tento uzel je v seznamu, tak ho do seznamu nepřidáme. Při vytváření předchůdců budeme nastavovat čítače v uzlech. Čítač nám říká, kolik hran vede z uzlu do splitru. Po vytvoření předchůdců vyresetujeme příznaky u všech přidaných uzlů. Vytvoříme kopii splitru, protože při zjemňování může dojit i k rozpadu samotného splitru. To by při následném zjemňování podle čítačů mohlo zapříčinit chybný rozpad bloků.



Obrázek 2.9. Vytvoření předchůdců splitru

Krok 4 Rozpad bloku

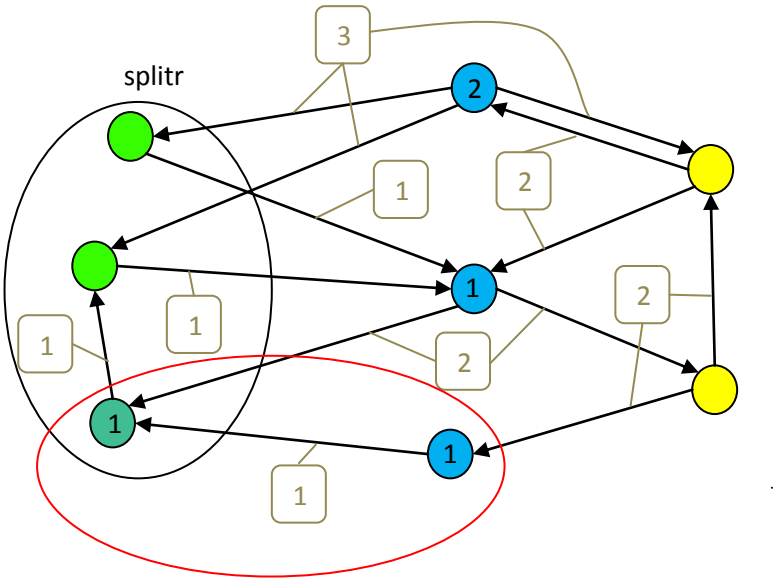
Provedeme rozklad každého bloku, ve kterém se vyskytuje předchůdce splitru. Blok se rozpadne na dvě části. V novém bloku jsou předchůdci splitru a v původním bloku zůstane zbytek uzlů. Může se stát, že všechny uzly bloku jsou předchůdci, v tomto případě zůstane původní blok stejný a nebude se rozpadat. Při rozpadu bloku musíme u nového bloku nastavit skupinu a nový blok do této skupiny přidat. Pokud skupina měla jen jeden blok, tak nebyla v seznamu se skupinami. Po rozpadu se skupina rozšířila o nový blok a má minimálně dva bloky, proto ji vložíme do seznamu se skupinami.



Obrázek 2.10. Rozpad bloku

Krok 5 Hledání uzlů, které mají pouze hrany vedoucí do splitru

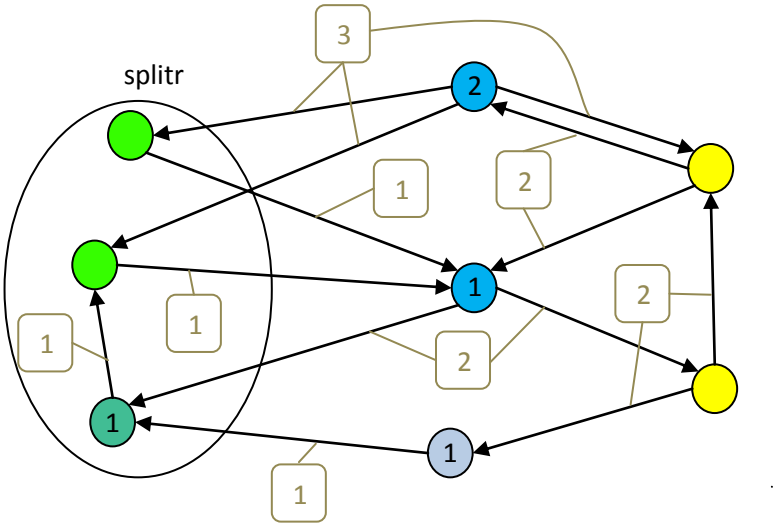
V předchozím kroku nám vznikly nové bloky obsahující uzly, které mají alespoň jednu hranu směřující do splitru. Tyto bloky obsahují i uzly, které mají hrany směřující pouze do splitru. Vyhledáme tyto uzly a pro pozdější rozpad bloku si je uchováme ve spojitém seznamu.



Obrázek 2.11. Uzly, které mají hrany vedoucí pouze do splitru

Krok 6 Rozpad nově vzniklých bloků.

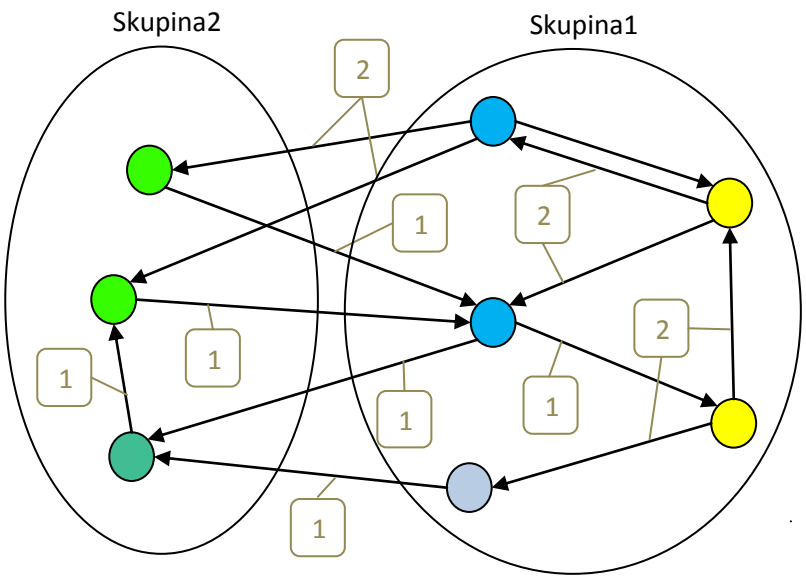
Je velmi podobný kroku 4, kde rozkládáme bloky podle předchůdců splitru. V tomto kroku rozkládáme nově vzniklé bloky podle uzlů, které jsme vyhledali v předchozím kroku. Provedeme rozklad každého bloku, ve kterém se vyskytují tyto uzly. V novém bloku jsou uzly, které mají hrany směřující pouze do splitru. V původním bloku zůstanou uzly, které mají alespoň jednu hranu, která nevede do splitru.



Obrázek 2.12. Rozpad nově vzniklých bloků

Krok 7 Nastavení čítačů

U každé hrany směřujících do splitru snížíme čítač o jeden a vytvoříme nový čítač pro uzel předchůdce. Hraně přiřadíme referenci na tento nový čítač. Jestliže z uzlu vede více hran do splitru, tak nevytváříme pro každou hranu nový čítač, ale s každou hranou čítač pro daný uzel zvýšíme a hraně přiřadíme referenci na tento čítač. Pokud po snížení má původní čítač nulovou hodnotu, tak jej smažeme. Po průchodu všech hran směřujících do splitru smažeme i kopii splitru.



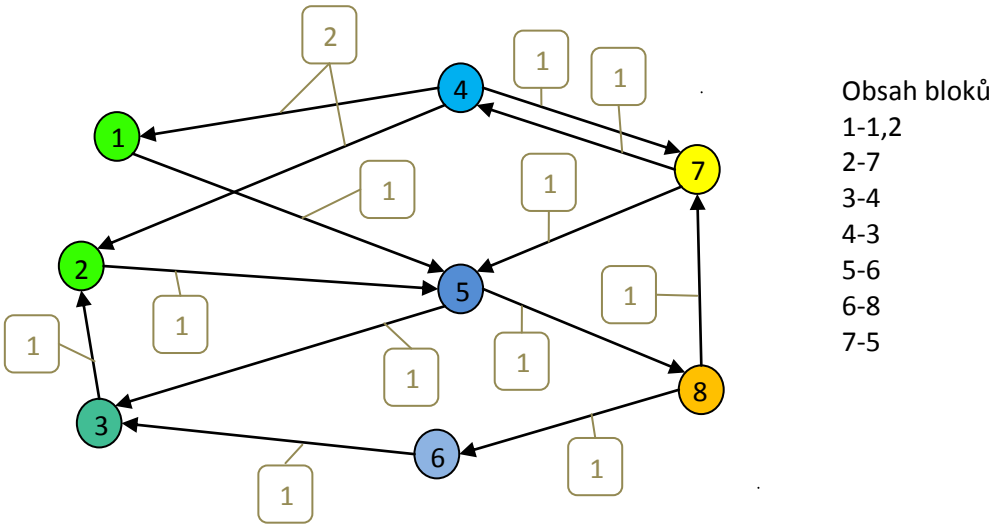
Obrázek 2.13. Přenastavení čítačů

Výstup

Na obrázku 2.14 máme konečný rozpad našeho příkladu a výpis obsahů bloků. Téměř všechny uzly patří do samotného bloku, až na uzly 1 a 2. Tyto dva uzly jsou ve společném bloku a jsou si ekvivalentní. Pokud bychom chtěli systém minimalizovat, tak bychom tyto dva uzly nahradili jedním.

Formát výstupu je:

Název bloku – název uzlu, název uzlu …



Obrázek 2.14. Konečný rozklad systému s textovým výstupem

3 Program Paige-Tarjan

V této kapitole se budeme zaobírat samotnou implementací algoritmu. Řekneme si, jaký formát vstupu program přijímá a seznámíme se s jednotlivými kroky algoritmu Paige-Tarjan. Rovněž si představíme grafické uživatelské rozhraní programu.

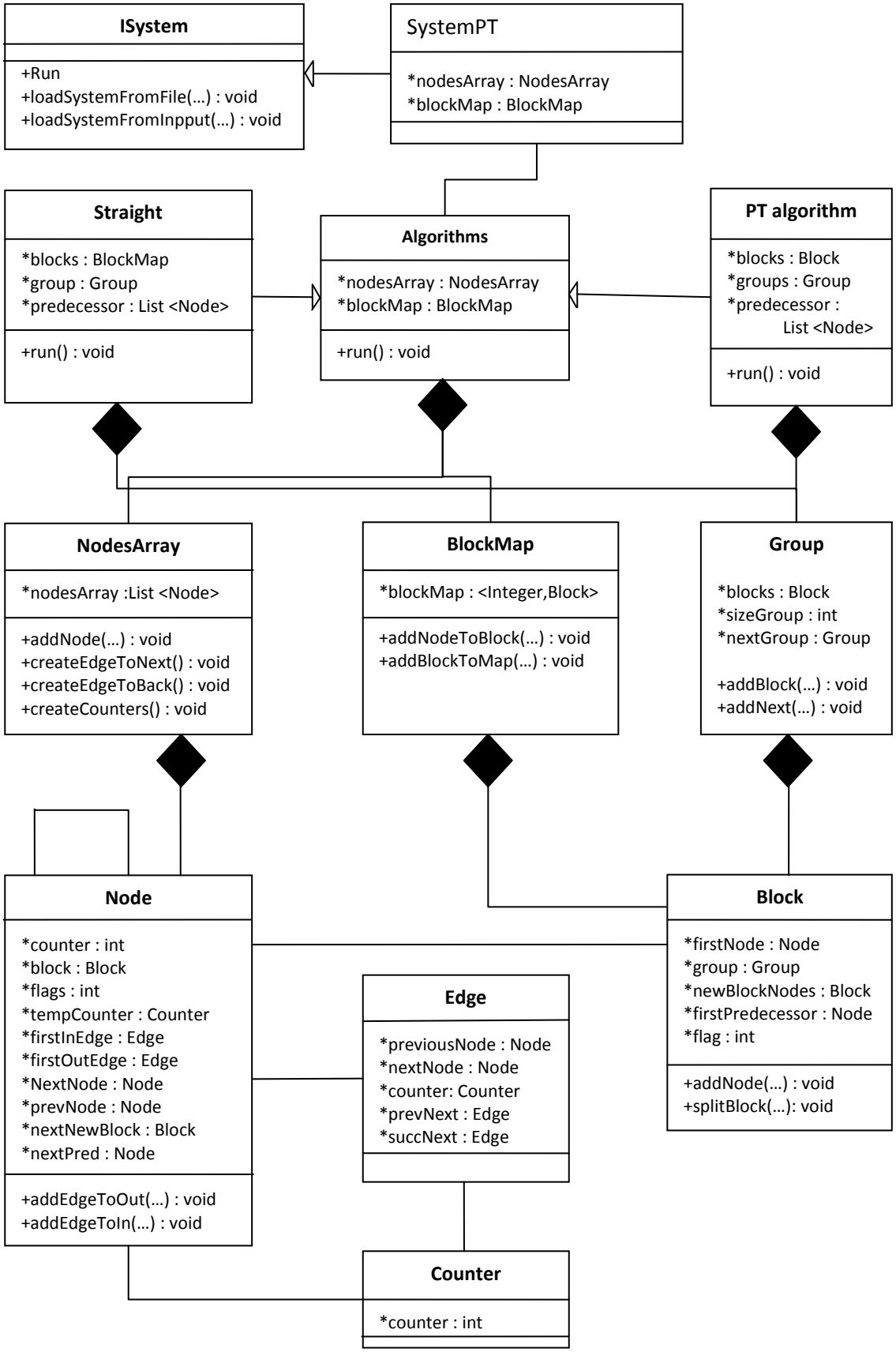
3.1 Třídní diagram programu

Strukturu programu si zobrazíme pomocí třídního diagramu viz obrázek 3.1.

Původně jsme spojitý seznam realizovali datovou strukturou linkedList, která je již v programovacím jazyku Java implementována. Jelikož metoda remove v linkedListu má složitost O (n), musíme si spojitý seznam naimplementovat sami, abychom docílili složitosti O (1). Realizaci tohoto spojitého seznamu si ukážeme na příkladu.

U bloku potřebujeme vytvořit spojitý obousměrný seznam, do kterého budeme vkládat uzly obsažené v bloku. Každý blok má odkaz na první uzel v seznamu, který pojmenujeme firstNode. Naopak každý uzel si udržuje odkazy na předchozí uzel (prevNode) a následující uzel (nextNode).

Na rozdíl od linkedListu, který si udržuje ve spojitém seznamu jen reference na uzly, tak v naší implementaci spojitého seznamu máme propojené samotné objekty. Podobným způsobem jsou realizovány spojité seznamy u ostatních objektů.



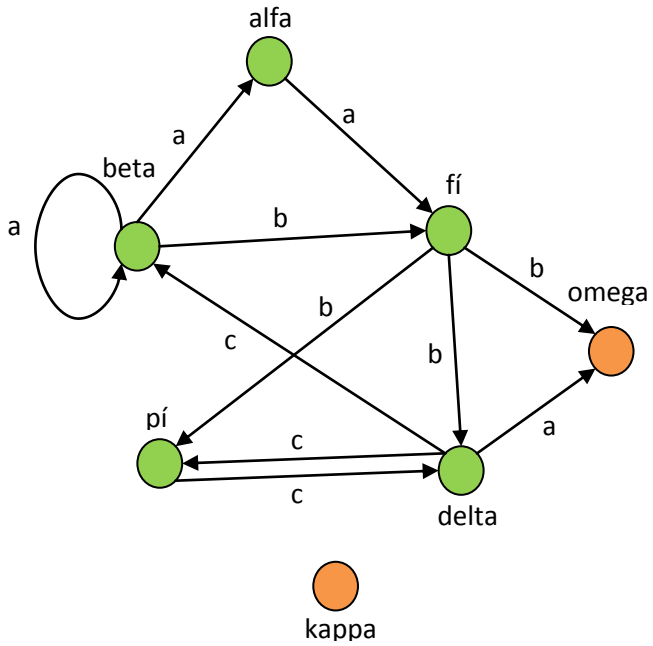
Obrázek 3.1. Třídní diagram algoritmu

3.2 Zadání a formát vstupů

Vstupem je orientovaný graf s pojmenovanými přechody a s uzly rozdělenými do bloků. Tento graf můžeme vyjádřit textovým zápisem, který bude mít následující strukturu. První slovo představuje název uzlu, za nímž je oddělovač v podobě pomlčky, poté následuje číslice označující blok a nakonec mezi složené závorky zapíšeme následníky v podobě řetězce. Při zápisu následníků nejdříve napíšeme pojmenování přechodu, poté vlnovkový oddělovač a pak název uzlu. Všichni následníci jsou od sebe vzájemně odděleni čárkou.

Formát vstupu:

Název uzlu-číslo bloku{název přechodu~název uzlu, název přechodu~název uzlu…}



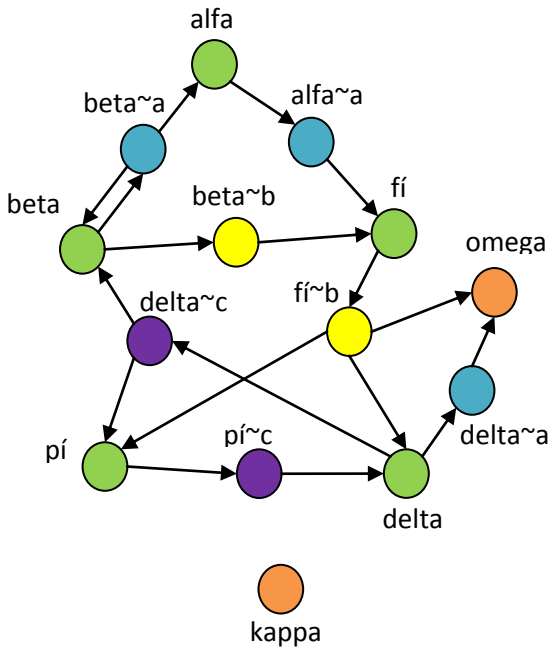
Alfa – 1 {a~fí}
Beta – 1 {a~alfa, a~beta, b~fí}
Fí – 1 {b~pí, b~delta, b~omega}
Pí – 1 {c~delta}
Delta – 1 {a~omega, c~beta, c~pí}
Kappa – 2 {}
Omega – 2 {}

Obrázek 3.2. Ohodnocený orientovaný systém s textovým zápisem

Orientovaný graf s pojmenovanými přechody je znázorněn viz obrázek 3.2 a vedle něj je uveden jeho textový zápis. Ten přetransformujeme jen na orientovaný graf již bez pojmenovaných přechodů. Textový zápis je stejný jako u orientovaného grafu s pojmenovanými přechody, ale u zápisu následníku odpadne název přechodu a vlnovkový oddělovač a píšeme pouze název následníka. Následníky od sebe oddělujeme čárkou. Přetransformovaný systém je znázorněn viz Obrázek 3.3 i s jeho textovým zápisem.

Zápis orientovaného grafu má formát:

Název uzlu-číslo bloku{ název uzlu, název uzlu…}



Alfa – 1 {alfa~a}
Beta – 1 {beta~a, beta~b}
Fí – 1 {fí~b}
Pí – 1 {pí~c}
Delta – 1 {delta~a, delta~c}
Omega – 2 {}
Kappa – 2{}
Beta~a – 3{alfa, beta}
Alfa~a – 3 {fí}
Delta~a – 3 {omega}
Beta~b – 4 {fí}
Fí~b – 4 {delta}
Delta~c – 5 {beta, pí}
Pí~c – 5 {delta}

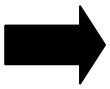
Obrázek 3.3. Orientovaný graf s textovým zápisem

Zápis orientovaného grafu je dosti složitý pro uložení do systému. V hodné je převést názvy uzlů na čísla. Vytvoříme list, kde budeme mít uloženy názvy uzlů a přidělené indexy. Index uzlů je pořadí, ve kterém načítáme uzly do listu, a když toto pořadí dodržíme i při vytváření základního vstupu, tak nemusíme uvádět index právě definovaného uzlu.

Základní vstup potom vytváříme tak, že každý řádek znamená nový uzel. Jelikož dodržujeme pořadí, tak první řádek v orientovaném vstupu je o uzlu s názvem *alfa*, tak v základním vstupu je první řádek taky *alfa*, druhý je *beta* atd. Tím nám odpadá psaní indexu definovaného uzlu a zapisujeme pouze číslo bloku, poté následuje oddělovací dvojtečka a za ní uvedeme indexy uzlů následníků. Následníky oddělíme mezi sebou čárkou.

Formát základního vstupu:

Číslo bloku:index následníka, index následníka …

Format orientovaného vstupu		**Základní formát vstupu**
Alfa – 1 {alfa~a}		1 : 9
Beta – 1 {beta~a, beta~b}		1 : 8,13
Fí – 1 {fí~b}		1 : 12
Pí – 1 {pí~c}		1 : 14
Delta – 1 {delta~a, delta~c}		1 : 10,13
Omega – 2 {}		2 :
Kappa – 2{}		2 :
Beta~a – 3{alfa, beta}		3 : 1,2
Alfa~a – 3 {fí}		3 : 3
Delta~a – 3 {omega}		3 : 6
Beta~b – 4 {fí}		4 : 3
Fí~b – 4 {delta}		4 : 5
Delta~c – 5 {beta, pí}		5 : 2,4
Pí~c – 5 {delta}		5 : 5

Obrázek 3.4 Základní formát vstupu

3.3 Transformace vstupního systému na základní formát

Algoritmus je schopen zpracovat vstupní systém jen v základním formátu. Zbylé dva formáty vstupního systému musíme přetransformovat do základního formátu.

O toto přetransformování se v programu stará třída *InputTransformation.* Při vytváření instance třídy *InputTransformation*, musíme v parametru zadat, o jaký vstup se jedná, abychom věděli, který textový formát budeme transformovat. Pochopitelně, pokud se jedná o základní formát, tak se nemusí nic transformovat.

Z textu nejdříve získáme názvy uzly. Každá definice uzlu končí symbolem „*}*“. Podle tohoto symbolu si rozsekáme vstup a vznikne nám pole, v němž máme uzly. Mezi názvem uzlu a číslem bloku se nachází rozdělovač „-“. Podle tohohle rozdělovače si rozložíme definici uzlu rozdělíme na dvě části. První část obsahuje název uzlu a druhá má číslo bloku a následníky. Tyto části si uložíme do mapy, kde první část je klíč a druhá je objektem. Pokud došlo k tomu, že se nám uzly nerozpadly na dvě části, tak vstup nebyl ve správném formátu a ukončíme transformování.

V tuto chvíli záleží, jaký je typ vstupu, jestli přetransformováváme ohodnocený orientovaný systém nebo jen orientovaný systém. Pokud je ohodnocený, tak si ho musíme převést na orientovaný systém. V mapě projdeme objekty, ve kterých je uložený blok a následníci. Z následníků získáme název přechodu. Název přechodu a následníci jsou rozděleni rozdělovačem „~“. Čili pokud je podle tohoto rozdělovače rozsekáme, tak nám vzniknout dvě části, první část je název přechodu a druhá část je název následníka. Vytvoříme nový uzel a jeho název se bude skládat z názvu původního uzlu, rozdělovače a názvu přechodu. Novému uzlu vložíme následníka, který je v druhé části. U původního uzlu musíme název přechodu s následníkem nahradit názvem nově vzniklého uzlu. Toto provedeme u všech následníku původního uzlu. Může se stát, že původní uzel má více přechodů se stejným pojmenováním. Nebudeme pro každý duplicitní přechod vytvářet nový uzel, využijeme už vytvořeného a přidáme mu dalšího následníka a u původního uzlu už nic nepřidáváme. V tuto chvíli máme v mapě uložené všechny uzly i s upravenými následníky a převedli jsme ohodnocený orientovaný systém na orientovaný systém. Nyní musíme přetransformovat orientovaný systém.

Názvy uzlů jsou uložené v mapě jako klíče, my tyto klíče uložíme do indexového seznamu, protože budeme potřebovat indexový přístup. Nyní je každý název uzlu pod určitým indexem. Objekty v mapě, ve kterých máme uložené bloky uzlů a následníky, jsou rozděleni rozdělovačem „{“. Podle tohoto rozdělovače rozdělíme řetězec na dvě části, na blok a následníky. Když už máme řetězec rozdělený, přetransformujeme ho na základní textový zápis uzlu. Zapíšeme číslo bloku z první části, poté ho oddělíme „:“ a místo názvu následníka zapíšeme jeho index, který zjistíme ze seznamu s názvy uzlů. Následníky od sebe oddělíme čárkou.

Tuto transformaci uděláme pro všechny objekty v mapě a každý takto přetransformovaný uzel zapíšeme na nový řádek. Vznikne nám základní vstup, kde každý uzel je na jednom řádku a číslo řádku odpovídá indexu v seznamu s názvy uzlů. Tento základní vstup uložíme do řetězcové proměnné s názvem b*asicInput* a v seznamu *nameNode* máme uložené názvy uzlů, které slouží ke zpětnému přetransformování. Základní vstup už dokáže systém načíst a provádět s ním potřebné operace.

3.4 Přípravné kroky před spuštěním algoritmu

Před samotným spuštěním algoritmu je potřeba udělat několik přípravných kroků. Musíme vytvořit všechny hrany, provést předzpracování vstupu, vytvořit čítače a všechny bloky vložit do prvotní skupiny.

Vytvoření hran

Každá hrana má reference na dva uzly, *prevNode a nextNode* (předchůdce a následníka). Uzel má dva seznamy s hranami. V jednom si udržuje hrany vedoucí z něj do jiných uzlů a ve druhém má hrany vedoucí do něj. Pokud hranu uložíme do seznamu s hranami vedoucí do následníků, tak reference *prevNode* je aktuální uzel a reference nextNode je uzel následníka. Hrany uložené ve druhém seznamu mají aktuální uzel v referenci nextNode a předchůdce v referenci *prevNode*.

Nejdříve si vytvoříme hrany vedoucí do následníků a poté až hrany vedoucí do předchůdců. Při vytváření hran vedoucích z uzlu budeme postupovat takto. U uzlu máme v seznamu *indexName* uložené indexy následníku. Ze seznamu všech uzlů získáme podle indexu uzel a vložíme ho hraně do *nextNode*. Do *prevNode* vložíme aktuální uzel.

Při vytváření hran vedoucích do uzlu postupujeme takto. Uzel už má naplněný seznam vedoucích z uzlu. Z tohoto seznamu získáme hranu a z hrany získáme následníka. Následníkovi vložíme tuto hranu do seznamu pro hrany směřující do uzlu.

Předzpracování

Nyní máme načtený systém a vytvořené všechny hrany, teď provedeme předzpracování. Projdeme všechny uzly a jejich následníky. Pokud najdeme uzel, který nemá následníky, projdeme všechny jeho předchůdce a v jejich následnících smažeme hranu směřující do uzlu bez následníků. Každý uzel patří do bloku, proto získáme jeho blok a tento uzel přesuneme do seznamu pro nový blok. Blok, ve kterém jsme provedli přesun našeho uzlu, uložíme do seznamu. Duplicitu v seznamu budeme řešit příznakem, který budeme před přidáním testovat. Označíme si každý zpracovaný uzel, abychom ho nezpracovávali při každém průchodu zbytečně vícekrát. Průchod seznamem s uzly provádíme do té doby, dokud nacházíme uzly bez následníků. Až průchod seznamem s uzly skončí, poté provedeme rozklad bloků, které máme uložené v seznamu.

Vytvoření čítačů

Čítač nám říká, kolik hran z jednoho uzlu vede do stejné skupiny. Vytvoříme čítače u hran. Jelikož na začátku algoritmu je jen jedna skupina, takže číslo v počítadle odpovídá počtu hran vedoucích z uzlu. Projdeme uzly a u každého uzlu vytvoříme čítač s číslicí, která označuje, kolik má uzel hran vedoucích do následníků. Potom u hran vedoucích do následníků vytvořím referenci na čítač.

Vytvoření prvotní skupiny

Na začátku algoritmu patří všechny bloky do jedné skupiny, ale postupně se bude skupina rozpadat na menší skupiny. Prvotní skupinu vytvoříme tak, že do ní vložíme všechny bloky. Prvotní skupinu vložíme do seznamu skupin.

3.5 Kroky algoritmu

Nyní máme systém připravený pro provedení samotných kroků algoritmu. Všechny uzly máme uloženy v seznamu *nodesArray*. U uzlu jsou vytvořeny oba seznamy s hranami. Blok má uložené uzly v obousměrném spojitém seznamu, předchůdce ve spojitém seznamu a má spojitý seznam pro rozpad bloku. Bloky jsou uloženy v mapě s názvem blocksMap. Skupina uchovává bloky ve spojitém seznamu. Skupiny, které mají více, jak dva bloky uložíme do spojitého seznamu. Hrana má referenci na počítadlo pod názvem counter. Každý uzel má v sobě čítač, ve kterém máme hodnotu určující, kolik hran vede do splitru.

Krok 1 výběr splitru

Ze seznamu skupin vyjmeme první skupinu. Z této skupiny vybereme první dva bloky. Z těchto dvou bloků vybereme ten, který má méně uzlů.

Krok 2 vytvoření nové skupiny pro spliter

Ze skupiny vyjmeme blok a vytvoříme pro něj novou skupinu. Vyjmutý blok vložíme do nové skupiny a u bloku nastavíme referenci na novou skupinu. Otestujeme skupinu, ze které jsme vybrali blok, jestli má alespoň dva bloky. Pokud skupina má alespoň dva bloky, tak ji vložíme zpátky do seznamu skupin. Vyjmutý blok budeme nazývat *splitr*.

Krok 3 vytvoření předchůdců splitru

Ve *splitru* projdeme všechny uzly a u každého uzlu projdeme seznam s hranami vedoucími do předchůdců. Předchůdce otestujem, jestli jsme ho už nepřidali do seznamu předchůdců bloku. Pokud ano, tak pouze u uzlu zvýšíme *counter*. Pokud ne, tak *counter* nastavíme na hodnotu jedna, vložíme uzel do seznamu předchůdců bloku a nastavíme u uzlu příznak, že jsme uzel přidali do seznamu předchůdců bloku. *Splitr* si zkopírujeme, protože při rozkládání můžeme rozložit i samotný *splitr*. Zkopírovaný *splitr* nazveme *temporarySplitr.*

Krok 4 rozpad bloků

Získáme předchůdce *splitru* a z něj získáme blok, do kterého uzel patří. Z bloku vyjmeme předchůdce a vložíme ho do seznamu pro rozpad bloku. Tento blok si uložíme do seznamu a duplicitu řešíme příznakem. Až toto provedeme pro všechny předchůdce, tak rozložíme bloky, které jsem si uchovávali v seznamu. Rozložení bloku provedeme tak, že nejdříve otestujeme, jestli seznam blok obsahuje alespoň jeden uzel. Pokud ne, tak přesuneme uzly ze seznamu pro rozpad bloku do bloku a nebudu blok rozkládat. Pokud ano, tak vytvoříme nový blok a do parametru *nameBlock* mu vložíme číslo o jedno větší než je počet všech bloků (číslo použijeme jako klíč pro uložení do mapy s bloky). Do obsahu nového bloku vložíme všechny uzly z původního bloku, které byli v seznamu pro rozpad bloku. V původním bloku vynulujeme seznam pro rozpad. Novému bloku nastavíme stejnou skupinu, jako měl původní blok a do skupiny vložíme nový blok. Pokud skupina měla jen jeden blok a teď se zvětšila na dva bloky, vložíme ji do seznamu skupin. Každý uzel si drží referenci na blok, do kterého patří, jelikož jsme přesunuly uzly z původního bloku do nového, tak musíme nastavit u převedených uzlů referenci na nový blok.

Krok 5 vyhledání uzlů pro rozpad podle čítačů

V *temporarySplitr* projdu uzly. U každého uzlu projdu hrany vedoucí do předchůdců. Porovnám *čítač* u hrany s *čitačem* u předchůdce a když se rovnají, tak získám blok, do kterého patří předchůdce. V získaném bloku přesuneme uzel z bloku do seznamu pro rozpad bloku a blok uložíme do seznamu, duplicitu řešíme příznakem.

Krok 6 rozpad bloků

Projdeme vytvořený seznam bloků a rozložíme je. Rozklad provádíme stejně jako v kroku 4.

Krok 7 reset čítačů

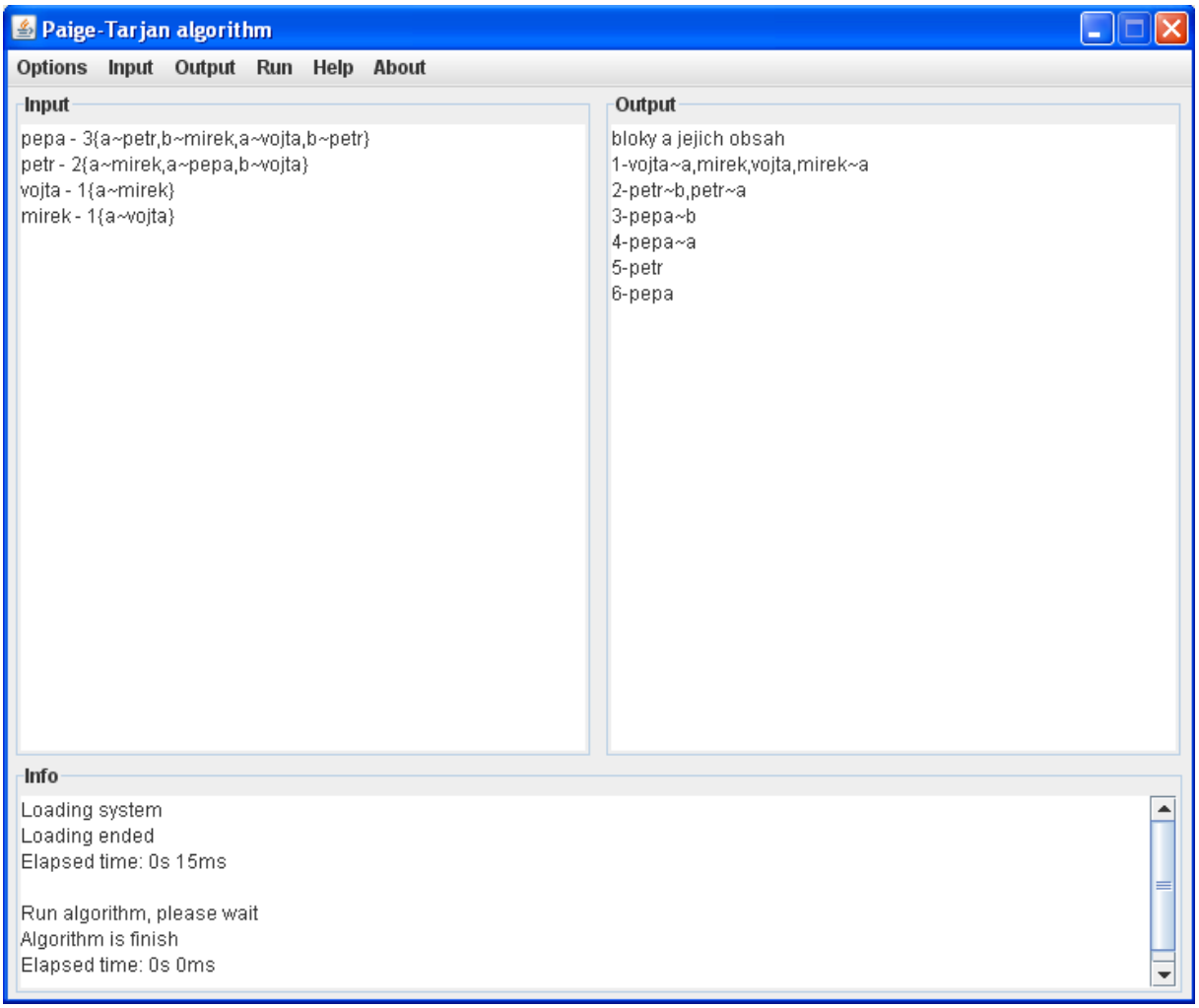
Ve *temporarySplitr* projdeme uzly a jeho seznam s hrany vedoucími do předchůdců. U každé hrany získáme čítač a snížíme ho o jeden. Po snížení čítače otestuji, jestli uzel referencí *temporaryCounter* ukazuje na nějaký čítač. Pokud ano, tak hraně do reference counter přiřadím ukazatel na tento čítač a čítač zvednu. Pokud ne, tak vytvořím nový čítač, hraně do reference *counter* přiřadím ukazatel na tento nový čítač a uzlu přiřadím ukazatel na tento čítač do *temporaryCounter*. U všech předchůdců *temporarySplitr nastavím* referenci *temporaryCounter* na null. Na konec smažu *temporarySpitr*

Výstup

Algoritmus skončí, až v seznamu skupin nejsou žádné další skupiny, ze kterých můžeme vybírat *splitry*.

Výstupem jsou čísla bloků a jejich obsah. V obsahu bloku jsou uvedeny jen indexy uzlů, a pokud vstup nebyl v základním formátu, tak tyto indexy musíme zpětně přetransformovat na názvy uzlů. Při přetransformování vstupu jsme si vytvořili seznam s názvem *nameNode*, kde jsme názvy uzlů uložili pod indexy, které používáme v algoritmu. Přetransformování výstupu provedeme tak, že podle indexu vyhledáme název uzlu v seznamu *nameNode* a při vypisování obsahu bloku vypíšeme místo indexu uzlu jeho název.

3.6 Popis grafického uživatelského rozhraní



Obrázek 3.5. Ukázka hlavního okna z programu

Okno je rozděleno do tří panelů. Panel *input* slouží k zapsání vstupu. V panelu *output* se nám zobrazí výsledek algoritmu a informace o aktuálním stavu programu se zobrazují v panelu *info* .
V menu options si můžeme nastavit formát vstupu, výstupu a zvolit si druh algoritmu, jež má být použit při zpracování vstupního systému.

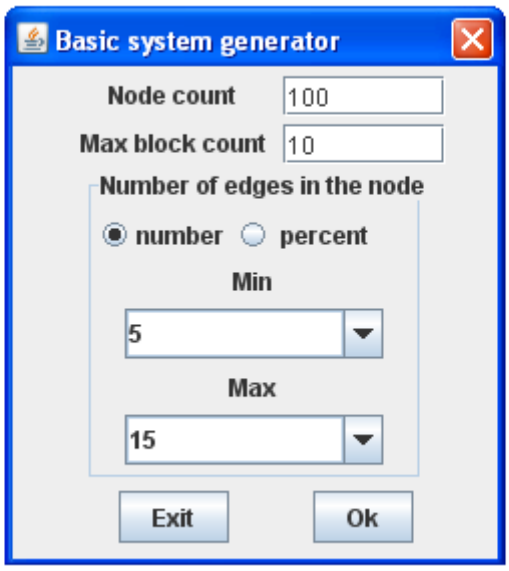
Rozlišujeme následující typy vstupního systému:

- Základní vstupní systém (*Basic*)
- Orientovaný graf (*Oriented*)
- Orientovaný graf s pojmenovanými přechody (*Labelled*)

Kromě zobrazení výsledku, si můžeme zobrazit samotný postup zpracování algoritmu pomocí volby debug.

Program umožňuje zpracovat vstupní systém pomocí algoritmu *straight* nebo algoritmu *Paige-Tarjan*.

Vstupní systém můžeme načíst do programu z externího souboru, nebo si jej můžeme vygenerovat pomocí volby *generate systém* viz obrázek 3.6. Vygenerovaný systém si můžeme uložit na disk v menu *output.*



Obrázek 3.6. Generátor náhodného vstupního systému

V první kolonce s popisem *node count* zadáváme počet uzlů systému. Počet bloků systému zadáváme v *max block count*. V panelu s názvem *number of edges in the node* nastavujeme počet hran u uzlu. Panel obsahuje dvě přepínací tlačítka *number* a *percent*. Těmito tlačítky volíme, jestli níže zadané hodnoty budou v procentech nebo je budeme brát jako čísla. Podle toho se nám i objeví nabídka u obou kolonek. V těch zadáváme rozsah počtu hran u uzlu. V *min* zadáváme nejnižší počet hran a v *max* zadáváme nejvyšší počet hran u uzlu.

Pomocí příkazu run zahájíme zpracování vstupní ho systému. Pro zpracování vstupních souborů větších než 100 Mb, zvolíme příkaz *run on file*. Vzhledem k tomu, že program není schopen takto velké soubory načíst do paměti, je systém zpracováván postupně a výsledek zpracování je zapsán do externího souboru.

4 Testování

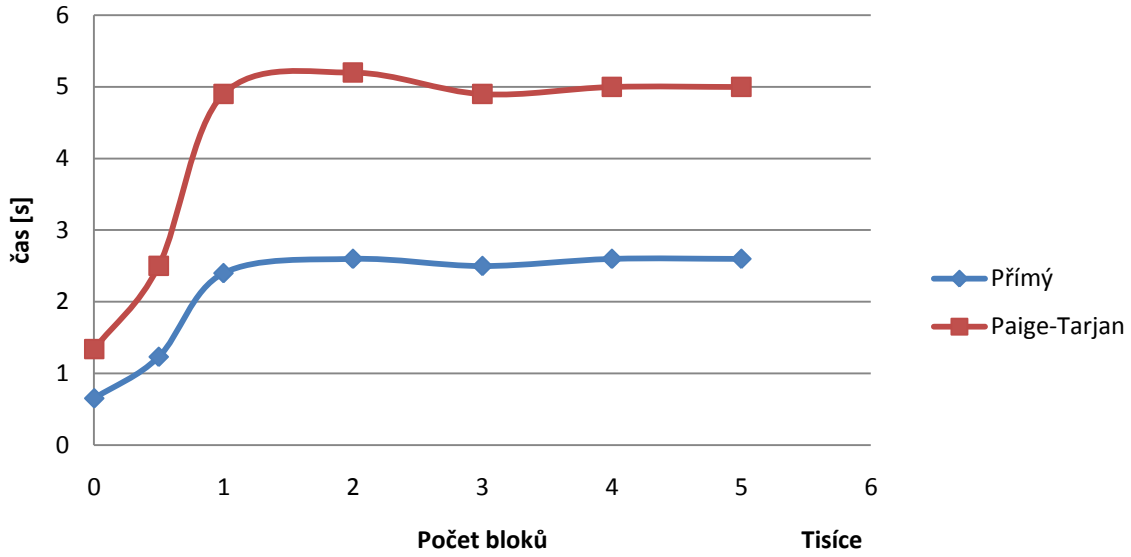
Z důvodu ověření správnosti výsledku jsem naimplementoval „přímý“ algoritmus. Nyní můžu provést srovnání efektivity obou algoritmů. Provedeme tři skupiny testů a v každém z nich budeme měnit jen jednu ze tří hlavních vlastností vstupu. Vstupy budeme vytvářet naším generátorem. U testů budeme měřit dobu zpracování vstupního systému.

4.1 Vliv změny počtu bloků na efektivnost algoritmu

V prvním typu testu jsme zvyšovali počet bloků, do kterých se nám rozpadne 50 000 uzlů. Z každého uzlu vede 50 hran směřujících do jiných uzlů. Výsledky měření jsme zapsaly do tabulky, ve které máme oba algoritmy. V prvním řádku se nachází počty bloků, pro které jsme provedli testy. Z výsledku měření jsme vytvořili graf v závislosti počtu bloků na čase.

Tabulka 1

	2	500	1000	2000	3000	4000	5000
Přímý	0,65	1,23	2,4	2,6	2,5	2,6	2,6
Paige-Tarjan	1,34	2,5	4,9	5,2	4,9	5,0	5,0



Graf 1. Závislost počtu bloků na čase

Z grafu vyplývá, že pokud bloky obsahují více uzlů, tak jsou algoritmy rychlejší. Úměra není lineární a existuje bod, při kterém už zvyšování počtu bloků nevede k výraznějšímu zpomalení algoritmů. V našem testu se tento bod nachází okolo 1000 bloků. Proč tomu tak je, se pokusíme vysvětlit na příkladě.

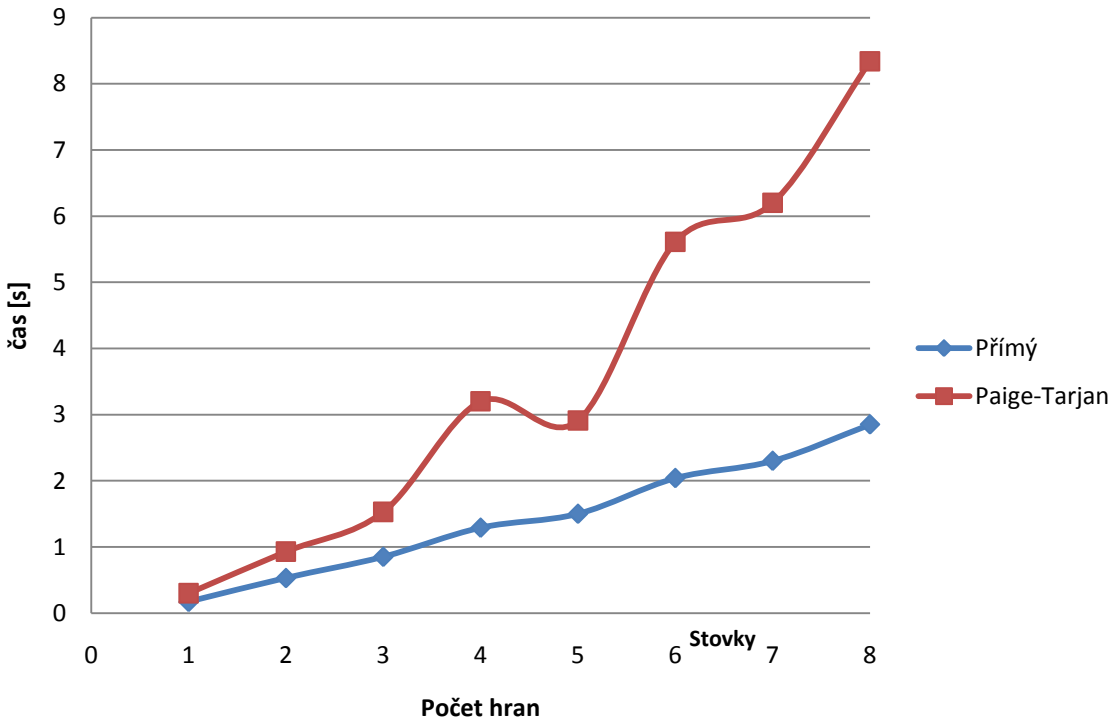
V předzpracování jsme vyhledávali uzly, které neměli hrany směřující do následníků a vytvářeli jsme pro ně nové bloky. Představme si, že máme takových uzlů třeba tisíc. Pokud tyto uzly budou ve dvou blocích, tak se nám systém rozpadne maximálně na 4 bloky. Pokud máme 1000 bloků a každý blok obsahuje jeden tento uzel, tak se nám systém rozpadne na 2000 tisíce bloků. Algoritmus musí navíc projít 1000 bloků, které nemusí vést k dalšímu rozpadu systému a ve výsledku to algoritmy zpomalí. V Prvním případě to byli pouze 2 bloky.

4.2 Vliv změny počtu hran u uzlů na efektivnost algoritmu

V tomto testu jsme postupně zvyšovali počet hran směřujících z uzlů, při zachování 5 000 uzlů rozdělených do 100 bloků. Výsledky měření jsme zapsaly do tabulek, ve kterých máme dva algoritmy. V prvním řádku se nachází počet hran, pro které jsme provedli testy. Z výsledku měření jsme vytvořili grafy v závislosti počtu uzlů na čase.

Tabulka 2

	0-100	100-200	200-300	300-400	400-500	500-600	600-700	700-800
Přímý	0,17	0,53	0,85	1,29	1,5	2,04	2,30	2,85
Paige-Tarjan	0,30	0,93	1,53	3,2	2,91	5,61	6,2	8,34



Graf 2. Závislost počtu hran u uzlů na čase

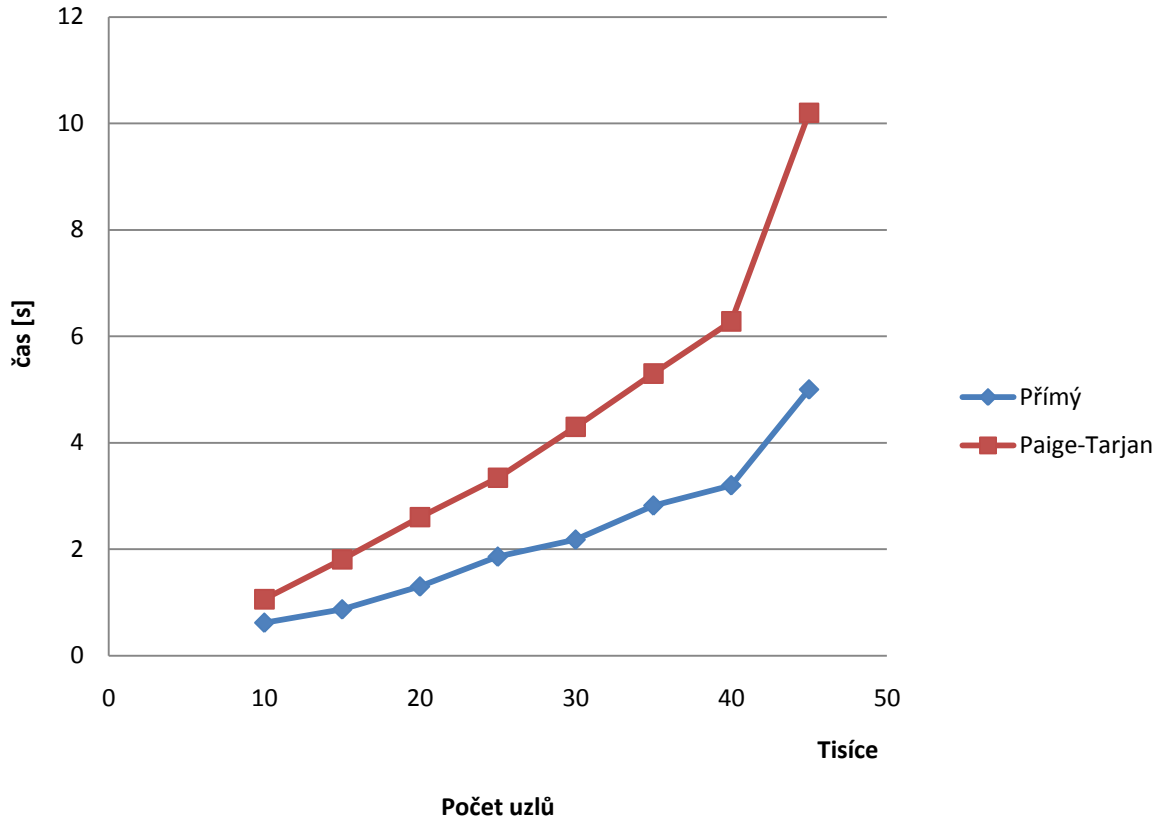
U obou algoritmů roste čas s přibývajícím počtem hran. Křivky nejsou úplně lineární, ale mírně se ohýbají do exponenciály. U algoritmu Paige-Tarjan je ohyb strmější, protože pro svůj běh spravuje více objektů, než přímý algoritmus.

4.3 Vliv změny počtu uzlů na efektivnost algoritmu

Při těchto testech jsme postupně zvyšovali počet uzlů. V prvním testu byly uzly rozděleny do 500 bloků a počet hran se bude v průměru pohybovat okolo 75 pro každý uzel. V druhém testu se počet hran pohyboval od 1 do 5 hran pro každý uzel a uzly byly rozděleny do 500 bloků.

Tabulka 3

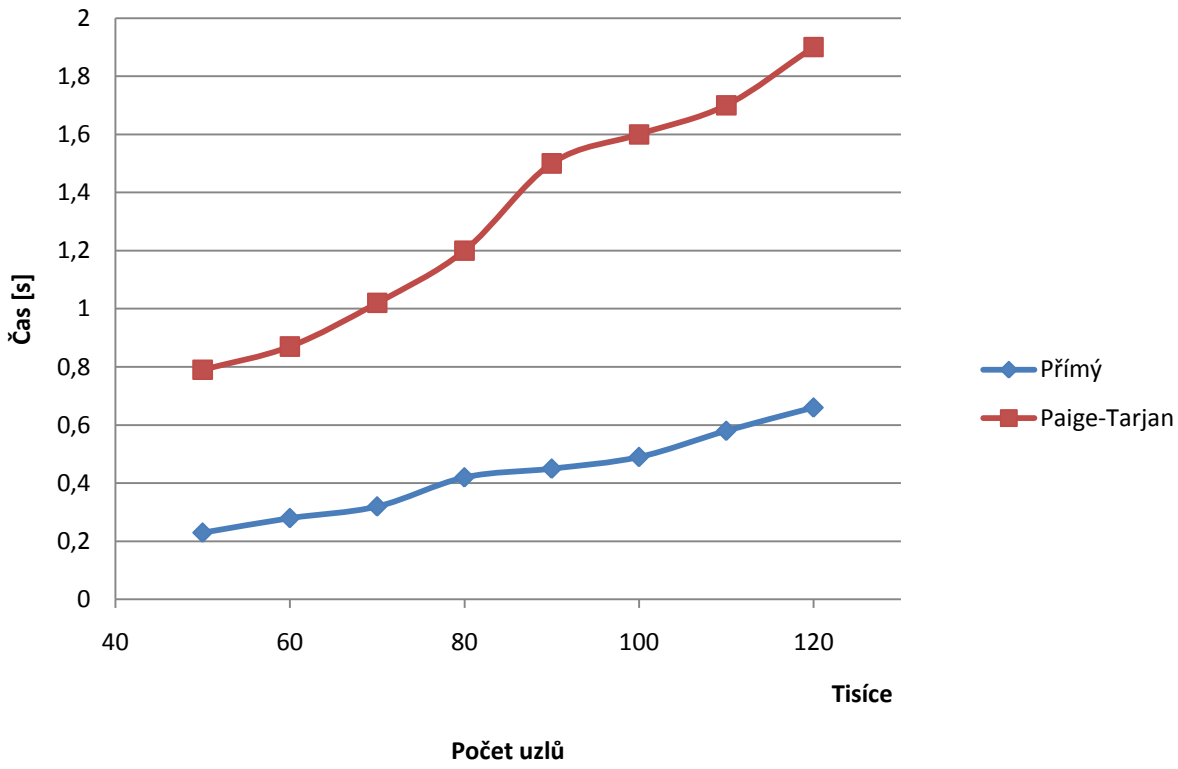
	10000	15000	20000	25000	30000	35000	40000	45000
Přímý	0,62	0,87	1,3	1,86	2,18	2,82	3,2	5
Paige-Tarjan	1,06	1,81	2,60	3,34	4,3	5,3	6,28	10,2



Graf 3. Závislost počtu uzlů na čase

Tabulka 4

	50000	60000	70000	80000	90000	100000	110000	120000
Přímý	0,23	0,28	0,32	0,42	0,45	0,49	0,58	0,66
Paige-Tarjan	0,79	0,87	1,02	1,20	1,5	1,6	1,7	1,9



Graf 4. Závislost počtu hran na čase

V obou grafech rostou křivky téměř lineárně, ale mají mírné zakřivení do exponenciály. Paige-Tarjan je opět pomalejší, protože potřebuje pro svůj běh spravovat více objektů.

5 Závěr

Ve své práci jsem naimplementoval algoritmus Paige-Tarjan. Abych si mohl ověřit správnost jeho výsledných rozkladů, tak jsem naimplementoval i „přímý“ algoritmus, který řeší stejný problém. Provedl jsem porovnání konečných rozkladů obou algoritmů pro stejná vstupní data. Ověření správnosti rozkladu záleží na tom, jestli uzly, které mají stejné chování, jsou ve společném bloku. Algoritmus Paige-Tarjan je korektně naimplementovaný, protože z hlediska ověření správnosti rozkladu jsou výsledky obou algoritmů shodné.

V testech vycházel přímý algoritmus lépe, než algoritmus Paige-Tarjan. Rozdíl v implementaci těchto dvou algoritmů je v tom, že Paige-Tarjan musí spravovat pro svůj běh čítače a skupiny. Tato správa objektů zabírá při běhu algoritmu určitý čas navíc, který přímý algoritmus neprovádí. Toto je jeden z důvodů, proč je přímý algoritmus rychlejší než algoritmus Paige-Tarjan. Dalším možným důvodem může být to, že algoritmus Paige-Tarjan je stavěný na nejhorší možný případ vstupu, který může nastat. Při testování jsem generoval náhodný vstup, který nemusel být nejhorším možným případem vstupu.

6 Literatura

[1] Robert Paige, Robert E. Tarjan: *Three partition refinement algorithms*, SIAM Journal on Computing, 16(6), pp. 973-989, Society for Industrial and Applied Mathematics, 1987

[2] Colin Stirling: *The Joys of Bisimulation,* Proceedings of MFCS'98: Symposium on Mathematical Foundations of Computer Science, Lecture Notes in Computer Science 1450, Springer-Verlag, 1998.

[3] Park, D: *Concurrency and automata on infinite sequences*, Lecture Notes in Computer Science, 154, 561-572, 1981.